



# Algoritmos de ordenamiento y búsqueda y sus complejidades de ejecución

Ingeniería en Computación e Informática



# Tabla de contenidos

- 1 Eficiencia de los algoritmos
- 2 Algoritmos de Ordenamiento
- 3 Algoritmos de Búsquedas

# Eficiencia de los algoritmos recursivos

El tiempo de ejecución de un algoritmo recursivo que usa la técnica *divide y conquista* viene dado por la suma de dos elementos:

- ▶ El tiempo que tarda en resolver los  $a$  subproblemas en los que se divide el original,  $a \cdot T(\frac{n}{b})$ , donde  $n/b$  es el tamaño de cada sub-problema.
- ▶ El tiempo necesario para combinar las soluciones de los sub-problemas para hallar la solución del original; normalmente es  $O(n^k)$ .

# Eficiencia de los algoritmos

Por tanto, el tiempo total es dado por la ecuación 1 de recurrencia.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^k) \quad (1)$$

La solución de esta ecuación, si  $a \geq 1, b \geq 2, k \geq 0$ , es:

- ▶ si  $a > b^k$ ,  $T(n) = O(n^{\log_b a})$
- ▶ si  $a = b^k$ ,  $T(n) = O(n^k \log n)$
- ▶ si  $a < b^k$ ,  $T(n) = O(n^k)$

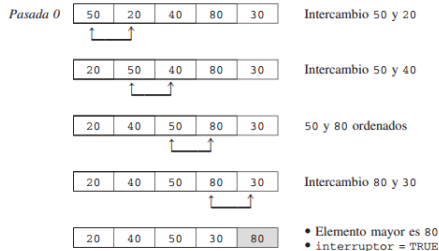
# Algoritmo de la Burbuja

- ▶ Se denomina ordenación por burbuja u ordenación por hundimiento debido a que los valores más pequeños *burbujean* gradualmente (suben) hacia la cima o parte superior del arreglo de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del arreglo.
- ▶ La técnica consiste en hacer varias pasadas a través del arreglo.
- ▶ En cada pasada, se comparan parejas sucesivas de elementos.
- ▶ Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están.
- ▶ Si una pareja está en orden decreciente, sus valores se intercambian en el arreglo.

# Algoritmo de la Burbuja

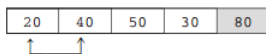
```
1 void burbuja (int *a, int n)
2 {
3     int i, j;
4     int temp;
5     int interruptor = 1;
6
7     /* pasadas */
8     for(i = 0; i < n && interruptor; i++)
9     {
10         interruptor = 0;
11         for(j = 0; j < n - i; j++)
12             if(a[j] > a[j+1])
13             {
14                 interruptor = 1;
15                 /* Intercambiamos */
16                 temp = a[j];
17                 a[j] = a[j+1];
18                 a[j+1] = temp;
19             }
20     }
21 }
```

►  $a = \{50, 20, 40, 80, 30\}$

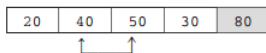


# Algoritmo de la Burbuja

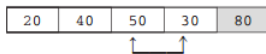
En la pasada 1:



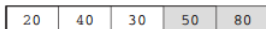
20 y 40 ordenados



40 y 50 ordenados

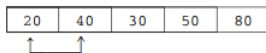


Se intercambian 50 y 30

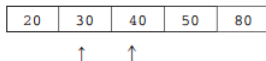


- 50 y 80 elementos mayores y ordenados
- interruptor = TRUE

En la pasada 2, sólo se hacen dos comparaciones:



20 y 40 ordenados




- Se intercambian 40 y 30
- interruptor = TRUE

# Algoritmo de la Burbuja

En la pasada 3, se hace una única comparación de 20 y 30, y no se produce intercambio:

20	30	40	50	80
----	----	----	----	----



20 y 30 ordenados

20	30	40	50	80
----	----	----	----	----

- Lista ordenada
- interruptor = FALSE



# Análisis del algoritmo de la burbuja

- ▶ ¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja?.
- ▶ Dependerá de la versión utilizada.
- ▶ En la versión más simple se hacen  $n - 1$  pasadas y  $n - 1$  comparaciones en cada pasada.
- ▶ Por consiguiente, el número de comparaciones es  $(n - 1) \cdot (n - 1) = n^2 - 2n + 1$ , es decir, la complejidad es  $\Theta(n^2)$ .

# Algoritmo de Selección

- ▶ De todos los elementos, se selecciona el elemento menor.
- ▶ Luego se intercambia el menor y se coloca en la posición correspondiente (dependiendo de la iteración en que se encuentre).
- ▶ Es como cuando tengo un grupo de cartas en la mano, selecciono la mejor y la coloco al inicio, luego con la siguiente y así sucesivamente.

```
1 void seleccion (int *a, int n)
2 {
3     int indiceMenor, i, j;
4     int aux;
5     /* ordenar a[0]... a[n-2] y a[n-1]
6        en cada pasada */
7     for (i = 0; i < n-1; i++)
8     {
9         /* comienzo de la exploración en
10            índice i */
11         indiceMenor = i;
12         /* j explora la sublista
13            a[i+1]..a[n-1] */
14         for (j = i+1; j < n; j++)
15             if (a[j] < a[indiceMenor])
16                 indiceMenor = j;
17         /* sitúa el elemento más pequeño
18            en a[i] */
19         if (i != indiceMenor)
20         {
21             aux = a[i];
22             a[i] = a[indiceMenor];
23             a[indiceMenor] = aux ;
24         }
25     }
26 }
```

# Algoritmo de Selección

A[0]	A[1]	A[2]	A[3]	A[4]
51	21	39	80	36

↓  
*pasada 0*

21	51	39	80	36
----	----	----	----	----

↓  
*pasada 1*

21	36	39	80	51
----	----	----	----	----

↓  
*pasada 2*

21	36	39	80	51
----	----	----	----	----

↓  
*pasada 3*

21	36	39	51	80
----	----	----	----	----

*Pasada 0.* Seleccionar 21  
Intercambiar 21 y A[0]

*Pasada 1.* Seleccionar 36  
Intercambiar 36 y A[1]

*Pasada 2.* Seleccionar 39  
Intercambiar 39 y A[2]

*Pasada 3.* Seleccionar 51  
Intercambiar 51 y A[3]

Lista ordenada

# Análisis del algoritmo de selección

- ▶ Al algoritmo de ordenamiento por selección, para ordenar un arreglo de  $n$  términos, tiene que realizar siempre el mismo número de comparaciones:

$$t(n) = \frac{n^2 - n}{2}$$

- ▶ Esto es, el número de comparaciones  $t(n)$  no depende del orden de los términos, si no del número de términos.

$$\Theta(t(n)) = \Theta(n^2)$$

# Algoritmo de Quicksort

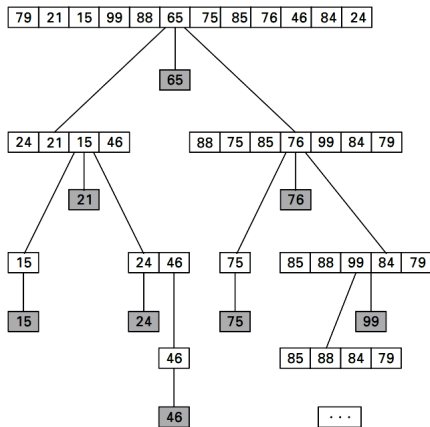
Es uno de los más eficientes para ordenar. Se basa en la división del problema en problemas cada vez mas pequeños, por tanto, es mucho mas natural describirlo como un procedimiento recursivo.

¿Como funciona?

- 1 Se escoge un elemento, que será nuestro pivote, al que llamaremos  $p$ .
- 2 Se divide el vector para que todos los elementos menores que  $p$  queden a su izquierda, y los elementos mayores que  $p$  queden a su derecha.
- 3 Se ordenan los sub-vectores delimitados por  $p$  utilizando el mismo proceso descrito.

# Algoritmo de Quicksort

```
1 void quicksort(double *a, int primero, int ultimo) {  
2     int i, j, central;  
3     double pivote;  
4     central = (primero + ultimo)/2;  
5     pivote = a[central];  
6     i = primero;  
7     j = ultimo;  
8     do {  
9         while (a[i] < pivote) i++;  
10        while (a[j] > pivote) j--;  
11        if (i <= j) {  
12            double tmp;  
13            tmp = a[i];  
14            a[i] = a[j];  
15            a[j] = tmp; /* intercambia a[i] con a[j] */  
16            i++;  
17            j--;  
18        }  
19    } while (i <= j);  
20    if (primero < j)  
21        quicksort(a, primero, j); /* mismo proceso con sublista izqda */  
22    if (i < ultimo)  
23        quicksort(a, i, ultimo); /* mismo proceso con sublista drcha */  
24 }
```



# Quicksort

## Complejidad

- ▶ Cada llamada genera dos llamadas recursivas  $a = 2$ .
- ▶ El tamaño del subproblema es la mitad del problema original  $b = 2$ .
- ▶ Sin considerar la recurrencia la operación restantes toman  $O(n)$  luego  $g(n)$  es  $O(n) = O(n^1)$ , ( $k = 1$ ).
- ▶ Para analizar este algoritmo, consideramos la ecuación 2 de recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad (2)$$

- ▶ Entonces  $a = 2$ ,  $b = 2$  y  $k = 1$ . Por ello,  $a = b^k$  y  $T(n) \in O(n \log n)$ .



# Comparación

## Algoritmos de Ordenación

Número	Selección	Burbuja	Quicksort
100	4 miliseg.	6 miliseg.	2 miliseg.
500	86 miliseg.	165 miliseg.	9 miliseg.
1000	0,3 seg.	0,7 seg.	18 miliseg.
2000	1,4 seg.	2,6 seg.	43 miliseg.
100000	1 hora	2 horas	3.4 seg.

# Búsqueda secuencial

- ▶ Los algoritmos de búsqueda secuencial y binaria son 2 de los algoritmos más usados para encontrar elementos en una estructura de datos.
- ▶ La búsqueda lineal probablemente es sencilla de implementar e intuitiva.
- ▶ Básicamente consiste en buscar de manera secuencial un elemento, es decir, preguntar si el elemento buscado es igual al primero, segundo, tercero y así sucesivamente hasta encontrar el deseado.
- ▶ Tiene una complejidad de  $O(n)$ .

# Algoritmo de Búsqueda secuencial

```
1 int busSecuencial(int *a, int n, int x)
2 {
3     int i;
4     for(i = 0; i < n; i++)
5         if (a[i] == x)
6             return i;
7     return -1; // no se encontró
8 }
```

# Búsqueda binaria

- ▶ Uno de los requisitos antes de ejecutar la búsqueda binaria, es que el conjunto de elementos debe estar ordenado.
- ▶ Funciona de la siguiente forma:
  - ▶ Si  $x$  es igual al elemento medio, termina. Si no:
  - ▶ **Dividir** el arreglo en dos subarreglos por la mitad. Si  $x$  es más pequeño que el elemento medio, elegimos el subarreglo izquierdo. Si  $x$  es mayor que el elemento medio, escogemos el subarreglo derecho.
  - ▶ **Conquistar**: solucionar el subarreglo determinando si  $x$  está en el subarreglo. A menos que el subarreglo sea lo suficiente pequeño, aplicar la recursión para lograrlo.
  - ▶ **Combinar**: Obtener la solución del arreglo a partir de la solución de los subarreglos.

# Búsqueda binaria

## Algoritmo

### ► versión iterativa

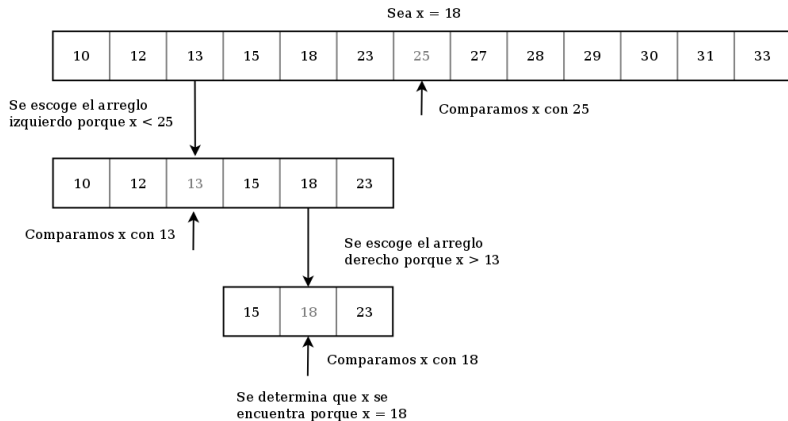
```
1 int busBinaria(int *a, int n, int x)
2 {
3     int i = 0;
4     int j = n-1;
5     while (i <= j)
6     {
7         int k = (i + j) / 2;
8         if (a[k] == x)
9             return k;
10        if (a[k] > x)
11            j = k - 1;
12        else
13            i = k + 1;
14    }
15    return -1; // No se encontró
16 }
```

### ► versión recursiva

```
1 int busBinaria(int *a, int x, int i, int j)
2 {
3     if (i > j)
4         return -1; // no se encontró
5     int k = (i + j) / 2;
6     if (a[k] == x)
7         return k;
8     if (a[k] > x)
9         return busBinariaRec(a, x, i, j-1);
10    else
11        return busBinariaRec(a, x, k+1, j);
12 }
```

# Búsqueda binaria

## Funcionamiento



# Análisis del algoritmo de búsqueda binaria

- ▶ Cada llamada genera una llamada recursiva  $a = 1$ .
- ▶ El tamaño del subproblema es la mitad del problema original  $b = 2$ .
- ▶ Sin considerar la recurrencia el resto de operaciones son  $O(1)$  luego  $g(n)$  es  $O(1) = O(n^0)$ , ( $k = 0$ ).
- ▶ Para analizar este algoritmo, consideramos la ecuación 3 de recurrencia:

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad (3)$$

- ▶ Entonces  $a = 1$ ,  $b = 2$  y  $k = 0$ . Por ello,  $a = b^k$  y  $T(n) \in O(\log n)$ .

# Comparación de las búsquedas

## Binaria y Secuencial

Tamaño del arreglo	Búsqueda binaria	Búsqueda secuencial
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000